IMA Journal of Mathematics in Management (1987) **1**, 277–291

A Type Calculus for Executable Modelling Languages†

GORDON H. BRADLEY AND ROBERT D. CLEMENCE, JR
Naval Postgraduate School, Monterey, California, USA

There is considerable current interest in the design and construction of directly executable modelling languages for mathematical programming. The present research extends contemporary modelling languages by specifying a type calculus for an extended dimensional system that determines if the model is well formed in the sense that the objective function and constraints consist of homogeneous components. Each variable, coefficient, constant, function, constraint, input, and output of the model is assigned a type that consists of its concepts, quantities, and units of measurement with optional scale factors. In checking the composition of functions and constraints, the system can do automatic unit conversions and apply user-supplied conversions. The system allows a hierarchy of concepts that provides inheritance of quantities and automatic concept conversion. In addition, each set in a model is typed so the system can check the validity of operations on indices.

1. Modelling and mathematical programming

AFTER many years of successful applications of mathematical programming to a wide variety of real-world problems, there is still a need to provide support that would make the development of models and construction of solutions faster and more reliable. There has been steady development with significant advances in solution algorithms for larger and larger problems; however, there has been less improvement in support for modelling and for the reliable construction of problems to be optimized.

The traditional, and still most widely used, approach is for the modeller to describe the model in an informal algebraic notation and then to develop a matrix generator computer program to construct the problem in the form required by an optimizer system. Although this approach has been, and is being, used successfully, it has several drawbacks that have prevented mathematical programming from being used as widely as it might be.

1. The development of matrix generators, even with the help of special-purpose languages, is a time-consuming and error-prone process.
2. The matrix generator process is difficult to validate because the output is voluminous and intended for machine processing, not human comprehension.
3. The relationship between the model and the matrix generator is often obscure and abstract, and thus it is difficult to determine whether the matrix generator corresponds to the modeller's intentions.
4. The model must be documented; in addition, the matrix generator and the relationship between model and generator must be documented.
5. Whenever a change is made in the model or in the data, the matrix

† Presented at the Martin Beale Memorial Symposium, Royal Society, London, UK, 6–8 July 1987.

© Oxford University Press 1988

generator must be modified. While changing the model may require hours, the modification and revalidation of the matrix generator may require days. This is particularly unsatisfactory when the model is undergoing constant revision, as in a planning application.

6. The informal notation for the model and the hand translation from model to matrix generator makes it impossible to provide automated help for many of the critical steps.

2. Executable modelling languages

An alternative to matrix generators is to create an executable modelling language (EML). This approach dispenses with intermediate forms altogether and makes the computer, not the modeller, responsible for the veracity of the model to optimizer translation. The concept is straightforward: the modeller conceives, records, and validates his or her model using a modelling language with algebraic notation that also documents the model. The model in this notation is read by a computer, translated into a form for optimization, solved, and its solution returned for analysis, all this without manual intervention.

The executable model approach requires two components: a modelling language and a modelling language translator. A modelling language is a declarative language that expresses the model in a notation that the computer can interpret; as such, it must satisfy two conflicting sets of requirements. First, it must be convenient for people; it must be easy to learn, easy to use, and as powerful and flexible as the informal algebraic notation it is intended to replace. Second, it must be understandable to a machine; it must have an unambiguous syntax and a notation compatible with ordinary computer hardware.

Our research extends contemporary executable modelling languages by defining a 'type' system for objects in the model. The type system can be added to any existing EML. This is illustrated below by the development of an example. For the example, typing is added to a 'generic' EML specified in Clemence (1987) that contains the features of several systems, for example those of Bisschop & Meeraus (1982), Clemence (1984), Day & Williams (1986), Fourer *et al.* (1987), Geoffrion (1986, 1987), Kendrick & Meeraus (1987), and Lucas & Mitra (1987). To demonstrate that typing can be added to all EMLs, the generic EML contains the significant features that interact with typing: for example, model and data separate (Clemence, 1984; Fourer *et al.*, 1987; Geoffrion, 1986, 1987) and stand-alone functions (Clemence, 1984; Geoffrion, 1986, 1987). We have omitted many of the valuable features like extensive set operations and conditional computations that interact less with typing and are not needed to demonstrate the power of typing.

Given a model description and data for the model, EML systems automatically construct a file that is ready for solution by an optimization system. Each particular data set, together with the model it corresponds to, forms an instance of the model that we will call a 'problem'. In most EML systems, the model and the data are in separate files. The separation of model and data is natural in situations where the model is developed and validated, and then many different

problems are formed by combining the model with different data files. In EML systems developed for planning use where the model can change as much as, or more than, the data, it may be more natural to intermix model and data in a single file. Even in systems with model and data mixed, it is useful to view the action of the systems as being composed of two components:

(i) an algebraic validation that determines if objects are correctly defined and if the functions, constraints, sets, and indices are composed with valid operations;

(ii) a data validation that determines if all the data is present and in the form required by the model.

In systems where model and data are separate, it is possible to include in the data the specification of sets (and thus their cardinality) as well as the value of coefficients. Changing the problem by changing only the data file offers significant power to the model user (who need not be the same person as the modeller). With this power goes the potential danger that data may not conform to the model requirements or to the modeller's intent. EML systems attempt to provide as much automatic support as possible to do data validation.

For contemporary EML systems, the algebraic validation is relatively weak. Although it is necessary to do the validation, most detectable errors are typographical and are easily corrected. The real substance of the modeller's intent is contained in the names of objects and in comments about the model components; while these are valuable as documentation, neither is subject to any mechanical validation. Data validation is similarly weak; only the documentation and the names of objects give the user any indication of the modeller's intent. If solution reports can be specified in the EML, then the statements can have algebraic validation, but here again there is no mechanism to establish an automatic link to the modeller's intent or to the data provided by the user.

3. Our contribution

Our research extends contemporary EMLs by specifying a type calculus for an extended dimensional system that determines if the model is well formed in the sense that functions and constraints consist of homogeneous components. Each variable, coefficient, constant, function, and constraint in the model is assigned a type that consists of its 'concepts', 'quantities', and 'units' of measurement with optional scale factors. This allows a powerful type validation that can automatically verify the modeller's intention to formulate consistent, meaningful functions and constraints. This automates what in contemporary EMLs are only comments requiring human validation.

The modeller assigns each numeric-valued object a type that consists of a concept description, quantity description, and unit description. The concept represents the essence of the object: examples are STEEL, TRUCKS, LABOUR HOURS, and PROFIT. A quantity is a measurable attribute of the concept: examples are WEIGHT, HEIGHT, CARDINALITY, and WEIGHT/LENGTH^2. A standard unit of measurement with an optional scale factor is specified for each

quantity. The units are from specified unit systems with conversion factors between units: examples are INCHES from ENGLISH LENGTH, [100]POUNDS from AVOIRDUPOIS WEIGHT, and POUNDS/INCHES from both. An example of a type is

WEIGHT of @BOX_OF_APPLES in POUNDS
quantity concept unit

Concepts are prefixed with the @ symbol to distinguish them clearly from quantities. Concepts are unchanged by multiplication and division, while the quantities and units are subject to the usual operations. For example, the WEIGHT of @STEEL in TONS divided by the VOLUME of @STEEL in $FEET^3$ yields WEIGHT/VOLUME of @STEEL in $TONS/FEET^3$.

During type validation, an EML can do automatic conversion of units, manipulate scale factors, and apply user-supplied concept, quantity, and unit conversions. The type system also allows a hierarchy of concepts, provides for inheritance of quantities, and supports automatic concept conversion. This allows the modeller to specify limits on how objects can be combined; compliance is automatically verified. This captures the modeller's intentions on how some objects in the model are to be used in functions and constraints.

Each of the sets used in defining the model is also assigned a type that controls the kind of operations that are allowed on the indices of the set. This reflects the modeller's intention about what the set represents and has implications about the ordering of members in the set.

Type validation also strengthens data validation. The user assigns a type to each number in the input data file and the system determines if it is consistent with the model. Automatic concept, quantity, and unit conversions are performed as necessary. A user can change the units or scale factors for the data and the system converts them to the required units. This protects the model from erroneous data and enforces the modeller's assumptions about the input data.

Type validation of solution report statements is also powerful. The concepts, quantities, and units of all numbers on the report can be specified. The system checks typing on all output, making automatic conversions as necessary. Model changes in the type of output quantities is either compensated for automatically, or, if that is not possible, an error is indicated.

Typing of objects need not be done entirely in the model. It is possible to make the type of objects part of the data. This makes it easy to add objects and modify objects in the data file without changing the model. It also allows the representation of classes of models (this is illustrated below).

Our research has focused on the design of a type calculus that is general enough to encompass all existing EMLs. Implementations of typing can be integrated into an EML or can be developed as a stand-alone type validation that follows the usual algebraic validation. The algorithms for implementing the type calculus are variations of the basic algorithms for doing lexical analysis, parsing, and evaluation of expressions, and are thus related to those in existing EMLs, for example those of Clemence (1984), Fourer *et al.* (1987), and Kendrick & Meeraus (1987).

4. Example using a typed executable modelling language

In this brief description of our work, we rely on the reader's intuition about dimensional systems to understand typing in the example. We have developed the notion of concept to add to classical dimensional systems that have only quantity and units. A complete syntax in BNF form and algorithms for operations on types are given by Clemence (1987). That work is based on three principles:

(i) the product or ratio of valid types is also a valid type;
(ii) for the operations of addition, subtraction, comparison ($\leqslant$, $=$, $\geqslant$), assignment, input, and output, both operands must have the same concept, quantity, and unit;
(iii) when simplifying expressions and considering automatic conversions, concepts are checked first, then quantities, and finally units.

The example is the shipment of butter measured in hundredweight per day from several dairies to warehouses. There is supply at each diary and demand at each warehouse. Given the costs of shipping from each dairy to every warehouse, the objective is to determine flows that will minimize shipping costs subject to the supplies and demands. The model expressed in the generic EML is shown in Fig. 1.

The typing information is contained within the symbols $\ll \cdots \gg$. The dimensional description, which consists of 'QUANTITY' of '@CONCEPT' terms, precedes the description of the units enclosed by $\# \cdots \#$ and includes an optional range. For variables, the range is interpreted as upper and lower bounds; for parameters, it specifies a data check on the range of input data. The indices (i, j) are not dummy, but are defined with the sets that they uniquely represent. The concept graph defines the relationships among the concepts in the model. Since the relationships are trivial for Fig. 1, the discussion of the concept graph is left until after the introduction of Fig. 3.

The basic concept in Fig. 1 is @BUTTER. The concept @BUTTER has the quantity WEIGHT_PER_PERIOD which is measured in 100's of pounds per day. The @OBJECTIVE concept has quantity COST measured in US dollars. The unit descriptions refer to the three unit systems which are included in the system. The unit system avoirdupois weight is measured in drams, ounces, pounds, and tons; standard time is measured in seconds, minutes, hours, days, and weeks; US currency is measured in pennies, nickels, dimes, quarters, halfdollars, and dollars.

The sets of DAIRIES and WAREHOUSES are typed nominal. The only operations allowed on the indices of such sets are equal, not equal, and membership. Index operations that involve ordering (NY < Boston), interval calculation (NY–Boston), and products or ratios ((NY–Boston)/2) are not permitted for nominal sets; thus, the user of the model knows that ordering is not important in specifying such sets in the data.

The system will recognize that the right- and left-hand sides of the INBOUND(j) constraints are in different units. Since both are in the avoirdupois weight system,

```
<< CONCEPT GRAPH
   @*   <--   @OBJECTIVE[COST]
   @*   <--   @BUTTER[WEIGHT_PER_PERIOD]>>

<< UNIT SYSTEMS
   WEIGHT_PER_PERIOD : AvoirdupoisWeight/StandardTime
   COST : USCurrency>>

SETS
   DAIRIES i; << nominal >>
   WAREHOUSES j; << nominal >>
   PATHS(i, j) := {DAIRIES} × {WAREHOUSES} ;

VARIABLES
   SHIPMENTS (i, j) {PATHS}; << WEIGHT_PER_PERIOD of @BUTTER
     # [100]POUNDS /DAY ; (0.0, ) # >>

PARAMETERS
   SCOST(i, j) {PATHS}; << COST of @OBJECTIVE/WEIGHT_PER_PERIOD of @BUTTER
     # DOLLARS/([100]POUNDS/DAY); (0.0, ) # >>
   SUPPLY(i) {DAIRIES}; << WEIGHT_PER_PERIOD of @BUTTER
     # [100]POUNDS/DAY; (0.0, ) # >>
   DEMAND(j) {WAREHOUSES}; << WEIGHT_PER_PERIOD of @BUTTER
     # TONS/WEEK; (0.0, ) # >>

FUNCTIONS
   OBJECTIVE := SUM (i, j) {PATHS} (SCOST(i, j) * SHIPMENTS(i, j));
     << COST of @OBJECTIVE; # DOLLARS; (0.0, ) # >>

CONSTRAINTS
   OUTBOUND(i) {DAIRIES} := SUM(j) {PATHS} (SHIPMENTS(i, j)) =L= SUPPLY(i);
     <<WEIGHT_PER_PERIOD @BUTTER # [100]POUNDS/DAY #>>
   INBOUND(j) {WAREHOUSES} := SUM(i) {PATHS} (SHIPMENTS(i, j)) =E= DEMAND(j);
     <<WEIGHT_PER_PERIOD @BUTTER # [100]POUNDS/DAY #>>

ASSERTION
   SUM(i) {DAIRIES} (SUPPLY(i)) =G= SUM(j) {WAREHOUSES} (DEMAND(j));

SOLVE
   MIN : OBJECTIVE; SUBJECT TO ALL;

REPORT
   PRINT:= SHIPMENTS (i, j) {PATHS}; << WEIGHT_PER_PERIOD of @BUTTER
     # [1000]POUNDS/DAY # >>
```

FIG. 1. Butter shipment model in typed EML.

this is not an error; the appropriate conversion will be applied before the file for the optimizer is generated.

The modeller has wide latitude to tailor the concepts and quantities to the particular situation. For instance, in the above example, the quantity WEIGHT_PER_PERIOD could be named SHIPPING_RATE. The concept @BUTTER could be modelled with two quantities WEIGHT in POUNDS and PERIOD in DAY. The type for SHIPMENTS(i, j) would then be WEIGHT/PERIOD of @BUTTER in

[100]POUNDS/DAY. An alternative would be to define two concepts each with a single quantity: WEIGHT of @BUTTER in POUNDS and PERIOD of @TIME in DAY. The above type would then be WEIGHT of @BUTTER/PERIOD of @TIME in [100]POUNDS/DAY.

This model is correct, but it does not fully represent the modeller's ideas in a formal way. The modeller intends a relationship between SCOST( ) and SHIPMENTS( ), namely that SCOST(i, j) applies to SHIPMENTS(i, j) and to no other. That is, SCOST(MI,NY) * SHIPMENTS(MI,BOSTON) is not valid. It is possible to define the concept @BUTTER(i, j), which means that @BUTTER(i, j) is a distinct concept for each (i, j) pair. Since @BUTTER(i, j) in the type of SHIPMENTS(i, j) cancels @BUTTER(i, j) in the type of SCOST(i, j), the objective function is still type valid and yields dollars. Invalid compositions are identified as errors.

Now that each flow has its own unique type, the constraints are not valid because they attempt to sum variables with different types. This could be rectified by applying a user-defined conversion TOB( ) that would convert @BUTTER(i, j) to @BUTTER:

OUTBOUND(i){DAIRIES} := SUM(j){PATHS}(TOB(SHIPMENTS(i, j))) =L= SUPPLY(i).

In addition to being awkward, this does not capture the modeller's intent that it is valid only to sum flows that all originate at the same diary or those that terminate at the same warehouse, and that all other combinations of flow are invalid. We now introduce the hierarchy of concepts that allows the modeller automatically to obtain exactly this limit on the combinations of flow. We introduce the concept @BUTTER(., j), which is butter that originates at any dairy and terminates at one specific warehouse j, and @BUTTER(i, .) that originates at dairy i and terminates in any warehouse. This idea is represented in the concept graph as shown in Fig. 2.

We interpret the up arrow as 'is a specialization of'. Thus @BUTTER(i, j) is an instance of @BUTTER(., j) or of @BUTTER(i, .). When the system encounters @BUTTER(MI,NY) + @BUTTER(MI,BOSTON), it automatically converts both to @BUTTER (MI,.) and adds. This conversion is irreversible: for example, 3 apples +4 oranges yields 7 fruit, but fruit can never be automatically converted to apples or oranges. In order for the constraints to be valid, each SUPPLY(i) must be typed @BUTTER(i, .) and each DEMAND(j) must be typed @BUTTER(., j). This, too, is what the modeller intended. By requiring that each data item be typed precisely, type validation augments data validation and assures that the data provided by

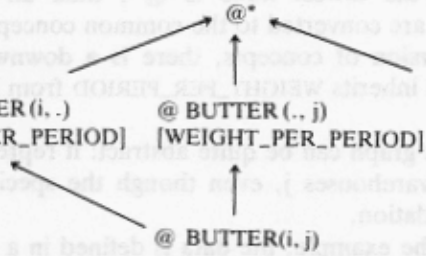

FIG. 2. Concept graph representation.

```
<< CONCEPT GRAPH
   @*                <--   @OBJECTIVE|COST|
   @*                <--   @BUTTER(i, .)|WEIGHT_PER_PERIOD|
   @*                <--   @BUTTER(., j)|WEIGHT_PER_PERIOD|
   @BUTTER(i, .)     <--   @BUTTER(i, j)
   @BUTTER(., j)     <--   @BUTTER(i, j) >>

VARIABLES
   SHIPMENTS (i, j) {PATHS};  << WEIGHT_PER_PERIOD of @BUTTER(i, j)
     # [100]POUNDS/DAY ; (0.0, ) # >>

PARAMETERS
   SCOST(i, j) {PATHS};  << COST of @OBJECTIVE/WEIGHT_PER_PERIOD of @BUTTER(i, j)
     # DOLLARS/([100]POUNDS/DAY); (0.0, ) # >>
   SUPPLY(i) {DAIRIES};  << WEIGHT_PER_PERIOD of @BUTTER(i, .)
     # [100]POUNDS/DAY; (0.0, ) # >>
   DEMAND(j) {WAREHOUSES};  << WEIGHT_PER_PERIOD of @BUTTER(., j)
     # TONS/WEEK; (0.0, ) # >>

CONSTRAINTS
   OUTBOUND(i) {DAIRIES} := SUM(j) {PATHS} (SHIPMENTS(i, j))  =L=  SUPPLY(i);
     <<WEIGHT_PER_PERIOD @BUTTER(i, .) # [100]POUNDS/DAY #>>
   INBOUND(j) {WAREHOUSES} := SUM(i) {PATHS} (SHIPMENTS(i, j))  =E=  DEMAND(j);
     <<WEIGHT_PER_PERIOD @BUTTER(., j) # [100]POUNDS/DAY #>>

REPORT
   PRINT:= SHIPMENTS (i, j) {PATHS};  <<  WEIGHT_PER_PERIOD of @BUTTER(i, j)
     # [1000]POUNDS/DAY # >>
```

FIG. 3. Changes to produce improved butter shipment model.

the user is consistent with the modeller's intentions. Figure 3 displays the sections of Fig. 1 that change to yield the improved model.

The root of the concept hierarchy is the universal type denoted as @*. Dimensionless quantities are assigned the universal type. Assigning all objects to the universal type has the effect of disabling the type validation. The concept hierarchy can be of any depth; butter can be converted to dairy product to foodstuff to freight in a model with some constraints specific to a particular food and others that deal with aggregations. When comparing two objects that must have the same type (e.g. objects to be added), if the concepts are not identical, then the system examines the concept graph to find the lowest concept that is 'up' from both concepts. If the lowest node is @*, then an error is declared; otherwise, both concepts are converted to the common concept. In addition to an upward automatic conversion of concepts, there is a downward inheritance of quantities. @BUTTER(i, j) inherits WEIGHT_PER_PERIOD from both @BUTTER(i, .) and @BUTTER(., j).

Notice that the concept graph can be quite abstract: it represents the hierarchy for all dairies i and all warehouses j, even though the specific sets will not be known until the data validation.

For the EML used in the example, the data is defined in a separate file that is similar to that used in Clemence (1984) and Geoffrion (1986). The sets SOURCE

and SINK are defined by listing the members. The ARC set is defined by listing the source and sink of each arc in the problem; thus, the arc set can be a subset of the Cartesian product of sources and sinks. Each number in the data file is assigned a type. When the data file is read, the system compares the type of each model coefficient with the type of its data file number. Automatic conversions are available: for instance, if the model has @FRUIT ← @APPLES in the concept graph and quantity conversion DENSITY ↔ WEIGHT/VOLUME, and if the type of a model coefficient is DENSITY of @FRUIT in POUNDS/FT^3 and the type of the corresponding data file number is WEIGHT/VOLUME of @APPLES in OZ/IN^3, then the types are found to be equivalent and the unit conversion is 108. The design of a flexible production quality data file with or without typing is challenging. If commercial database systems incorporated the typing of elements as described here, there would be no need for modelling systems to include a data file component.

5. Problem classes and application domains

The reader has undoubtedly noticed that the example is a transportation model, or, more precisely, it is an instance of a class of optimization models called transportation models. What is the 'transportationess' of this model that allows us to declare it an instance of an abstract class of models? The type system provides powerful automatic checks to determine if a particular data file can generate a valid instance of a model; what aid can it provide in determining if a specific model is an instance of a class of models?

Abstracting slightly from the example, we can get the class of transportation models specified in Fig. 4.

PROPOSITION 1 *A model is in the class of transportation models if and only if it is equivalent to Fig.* 4, *that is, it can be formed by substituting names, concepts, quantities, units, and unit systems.*

This is what we mean by the abstraction 'transportation model'. In Bradley & Clemence (1987) and Clemence (1987), an EML with typing is used to construct, from distinct models, a more comprehensive 'integrated' model. One topic discussed in these works is the use of libraries of validated model classes to be incorporated into larger models using a special library statement based on the idea in Proposition 1.

Although the proposition specifies the class of transportation models, it does not use the real power of the typing system, nor does it adequately capture the intention of the modeller. A set of conditions that gives necessary but not sufficient conditions for a model to be a transportation model seems to capture the essence of the role of model classes more clearly, and this gives a more constructive view of a class of models (see Fig. 5).

PROPOSITION 2 *Any model that is in the class of transportation models must incorporate a kernel model that is equivalent to Fig.* 5.

Figure 5 is a more abstract and more general definition of the class of transportation problems. Like Fig. 4, it specifies a single commodity and a

```
<< CONCEPT GRAPH
  @*                    <--  @Goal[GoalQuantity]
  @*                    <--  @Commodity(i, .)[CommodityQuantity]
  @*                    <--  @Commodity(., j)[CommodityQuantity]
  @Commodity(i, .)      <--  @Commodity(i, j)
  @Commodity(., j)      <--  @Commodity(i, j) >>

<< UNIT SYSTEMS
  CommodityQuantity : CommodityQuantityUnitSystem/TimeQuantityUnit System
  GoalQuantity : GoalQuantityUnitSystem>>

SETS
  Sources i;  << nominal >>
  Sinks j;  << nominal >>
  Arcs(i, j) := {Sources} x {Sinks} ;

VARIABLES
  Flow(i, j) {Arcs}; << CommodityQuantity of @Commodity(i, j)
    # CommodityQuantityUnit/TimeQuantityUnit ; (0.0,  ) # >>

PARAMETERS
  FlowCost(i, j) {Arcs}; << GoalQuantity of @Goal/CommodityQuantity of @Commodity(i, j)
    # GoalQuantityUnit/(CommodityQuantityUnit/TimeQuantityUnit); (0.0,  ) # >>
  Supply(i) {Sources}; << CommodityQuantity of @Commodity(i, .)
    # CommodityQuantityUnit/TimeQuantityUnit; (0.0,  ) # >>
  Demand(j) {Sinks}; << CommodityQuantity of @Commodity(., j)
    # CommodityQuantityUnit/TimeQuantityUnit; (0.0, ) # >>

FUNCTIONS
  Objective := SUM(i, j) {Arcs} (FlowCost(i, j) * Flow(i, j));
    <<GoalQuantity@Goal; # GoalQuantityUnit; (0.0, ) # >>

CONSTRAINTS
  Outbound(i) {Sources} := SUM(j) {Arcs} (Flow(i, j)) =L= Supply(i);
    <<CommodityQuantity@Commodity(i, .) # CommodityQuantityUnit/TimeQuantityUnit # >>
  Inbound(j) {Sinks} := SUM(i) {Arcs} (Flow(i, j)) =E= Demand(j);
    <<CommodityQuantity@Commodity(., j) # CommodityQuantityUnit/TimeQuantityUnit # >>

ASSERTION
  SUM(i) {Sources} (Supply(i))  =G=  SUM(j) {Sinks} (Demand(j));

SOLVE
  MIN : Objective; SUBJECT TO ALL;

REPORT
  PRINT:= Flow(i, j) {Arcs}; << CommodityQuantity of @Commodity(i, j)
    # CommodityQuantityUnit/TimeQuantityUnit # >>
```

FIG. 4. Class of transportation models.

bipartite graph; but unlike Fig. 4, it includes models where OUTBOUND equals SUPPLY, models where INBOUND is greater than or equal to DEMAND, and models where SUPPLY equals DEMAND. On the other hand, there are completions of Fig. 5 that are not transportation models. Ultimately it is a matter of judgement as to what defines a class of models; it is the authors' opinion that Fig. 4 is too restrictive and that Fig. 5 better characterizes the essence of the class.

```
<< CONCEPT GRAPH
@*              <--   @Goal[GoalQuantity]
@*              <--   @Commodity(i, .)[CommodityQuantity]
@*              <--   @Commodity(., j)[CommodityQuantity]
@Commodity(i, .)  <--   @Commodity(i, j)
@Commodity(., j)  <--   @Commodity(i, j) >>

<< UNIT SYSTEMS
CommodityQuantity : CommodityQuantityUnitSystem/TimeQuantityUnitSystem
GoalQuantity : GoalQuantityUnitSystem>>

SETS
Sources i;  << nominal >>
Sinks j;  << nominal >>
Arcs(i, j) := {Sources} × {Sinks} ;
```

FIG. 5. Kernel model for class of transportation models.

One way to think about Proposition 2 is to imagine that there is a 'master modeller' who constructs Fig. 5 and makes it available to a modeller. If the modeller builds a model incorporating Fig. 5 that fails type validation, then, by Proposition 2, it is not a transportation model. The idea is that a master modeller studies a class of problems closely and then develops a kernel model (like, for example, Fig. 5) that specifies the critical aspects of the class. The modeller with far less experience can then use the kernel model to build a specific model. The typing system, through type validation, can identify any aspect of the model that violates the necessary conditions for a valid problem in the class. The typing system acts as a surrogate for the master modeller as it looks over the shoulder of the modeller pointing out errors.†

In addition to industry use, the typing system could also help teach modelling skills. A class assignment could be to build a model for a problem using a kernel model like Fig. 5. The student would get immediate feedback from the typing system on the EML if the proposed solution violated any of the important properties of the class of models. Although perhaps not replacing the current teaching of modelling by studying examples, the EML with typing allows modelling practice with an automated safety net, and helps the student and teacher abstract the important properties that characterize classes of models.

Another important area for the master modeller and kernel models is to define an application domain. Here, the master modeller is assumed to be an expert in an application domain‡ like banking or mobilization of troops or manufacture of cars. The master modeller identifies the concepts, quantities, unit systems, conversions, concept graphs, etc. that are central to the application. A kernel model is constructed that is general enough to cover most of the valid models that

† Researchers in artificial intelligence might call this an 'expert system'; we prefer the term typing system since it has other uses and benefits.

‡ Our use of the term domain is consistent with its use in the cognitive sciences. For example, '. . .the psychological concept of knowledge domains. A knowledge domain consists of a closed set of primitive objects, properties of the objects, relations among objects and operators which manipulate these properties or relations.' (Brooks, 1978).

could be built for the application, but yet limited enough that violation of any important properties or relationship would be caught by the typing system. This is then given to a modeller who will generally be knowledgeable about his or her specific problem but not an expert on all modelling problems in the domain. The typing system of the EML will detect any violations of the properties the master modeller thought important for the domain.

An example of an application domain is banking. The master modeller might develop a unit system for time that includes the normal banking assumption that all months have 30 days and all years 360 days. There might be unit systems for each major currency and type conversions for exchange rates; an alternative would be a single money system with conversions built into the unit system. The concepts of the domain might be bonds, mortgages, and other financial instruments. The typing system would distinguish between money in different time periods and would allow money in Period t_1 to be added to money in Period t_2 only after the proper conversion using the discount rate. The concepts of money in different periods could also be built into a concept graph so that moneys in different periods would be automatically converted to present value. Banking conventions, assumptions about the domain, and other important features could be specified in the kernel model. A banker with limited modelling experience could model using this kernel domain model, and the typing system would enforce the master modeller's view of the domain. When kernel domain models are used in this way, the typing system becomes a method for abstracting and codifying the application domain and provides an automatic means to determine if a specific model is a valid instance of models from that domain.

The ease of use and quick construction of models and problems with EMLs, together with the automatic validation of application domain properties provided by a typing system, offers a significant opportunity to expand greatly the number of people who can effectively use optimization models.

6. Additional features of a typed EML

The transportation example does not demonstrate all the features of a typed EML that are developed in Clemence (1987). For instance, the example did not demonstrate that each constant in the model must be assigned a type. Concepts can have more than one quantity, for example WEIGHT, COST, HEIGHT, and SHELFLIFE, each with a unit of measurement. Conversions like our TOB( ) are an important part of typed models.

Simple extensions of our example also require some additions. For transshipment nodes k that involve a sum of @BUTTER(i, k) and @BUTTER(k, j), the concept hierarchy needs @BUTTER(*, k) which means either (k, .) or (., k). Gains on the transportation arcs, GAIN(i, j)*SHIPMENTS(i, j), demonstrates a typical use of parameters in a type system. GAIN(i, j) has concept @BUTTER(., j)/@BUTTER(i, j) so it converts (i, j) flow to (., j) flow. This is common in production models where typing shows that the product of a parameter and a variable represents the transformation of one product to another.

In addition to nominal sets, typing allows ordinal sets (nominal plus ordering of elements), interval sets (ordinal plus differences of elements), and ratio sets (interval plus products and ratios of elements). This allows the master modeller to place restrictions on the modeller's use of index operations. Although this mechanism cannot resolve all problems that arise in index manipulations and specification of sets in the data file, it does allow automatic checks and encourages communication among master modeller, modeller, and user on the use and misuse of indices.

The example shows the use of range specifications, e.g. (0-0,), within # · · · # to bound variables or to check parameters. It is also possible to compute the numerical precision of the computations to construct the optimization problem and to do interval arithmetic for the computations. If the number of digits of precision is given for all parameters, it is possible to propagate the precision of the calculations using algorithms that are minor modifications of those that propagate type. Similarly, given upper, lower, and most likely values for numbers, it is possible to propagate this interval information.

The example shows that type information can be abstract (the elements of the sets will not be specified until the data is read) and can involve Cartesian products of linear sets. There is a different type for each element defined over PATHS(i, j). Type validation can be completed even on this abstract level. There are examples where type validation cannot be completed until the data is read. For example, in a production problem, there might be a set of PRODUCTS(i) that will be specified in the data. A construct PRODUCT(A) + PRODUCT(B) cannot be type validated until the data file with the type of each is read. If type validation cannot be completed until the data is read, we call it 'partial' type validation and it generates assertions about the data that must be verified in order to achieve type validation, for example

ASSERTION: type (PRODUCT(A)) =?= type (PRODUCT(B)).

The ASSERTION mechanism, which naturally generalizes to

ASSERTION condition THEN action1 ELSE action2,

is a powerful mechanism for providing additional protection. However, in all its variations, it is difficult to implement, and it is a yet unresolved implementation decision whether the benefit of a full implementation is worth the cost.

As discussed above, some type information can be part of the data file. It has not escaped our attention that an extension of this would allow significantly more of the model to become data. For example, extending what can be data could allow Fig. 4 to be the model and the specializations that yield Fig. 3 could be in the data file. Although this is technically feasible, we believe the hierarchy of expertise, master modeller/modeller/user, and the protection in depth provided by kernel model/model/problem is the better approach that is more likely to achieve the goal of getting mathematical programming models to be used more widely.

7. Conclusions

We believe that the type calculus for executable modelling languages outlined here advances the goal of making model construction, validation, and interpretation easier, faster, and more reliable. The hierarchy of expertise and protection in depth provides new validation capability for modelling.

The concept of kernel domain model and classes of models is critical to the power of a typed EML. Only the modeller can map real objects into model objects and only the modeller can validate the relationship between reality and model. But a typed EML, together with a kernel model, provides a context in which there is significant automatic support to check that the modeller correctly manipulates the objects he or she defines. Type and concept hierarchy information is critical. Contemporary EMLs without typing have only the universal context of the properties of real numbers and the laws of algebra: this is too weak to provide any significant verification that the model captures the modeller's intent.

In developing typing, some ideas with surprising power (to us) emerged. First, concepts and the calculus to manipulate them are important and should be added to classical dimensional descriptions. Second, typing of objects with multiple indices can be done quite naturally and with as much abstraction as humans employ. Third, some part of the human capacity to effortlessly generalize and specialize typing information can be captured with a rather simple notion of concept graph. And, finally, typing adds a new dimension to our representation of classes of models. It seems that human understanding about classes of models has always been heavily based on our intuition about typing and type operations: making this explicit helps us to understand some characteristics of model classes. In short, these four points suggest that many intuitive ideas that humans use to think about models can be formalized in a very natural way, and making this explicit adds to our understanding of models and the modelling process.

Acknowledgements

The first author would like to acknowledge the support of the Mathematics Group of the Office of Naval Research. The authors would like to thank Art Geoffrion and Dan Dolk for introducing them to executable modelling languages. The authors also appreciate the comments and encouragement of Alex Meeraus, Bruce MacLennan, and Richard Hamming.

References

Bisschop, J., & Meeraus, A. 1982 On the development of a general algebraic modeling system in a strategic planning environment. *Math. Program. Stud.* **20,** 1–29.

Brooks, R. 1978 Using a behavioral theory of program comprehension in software engineering. *Proc. 3rd Int. Conf. Software Engineering*, pp. 196–201.

Bradley, G. H., & Clemence, Jr, R. D. 1988 Model integration with a typed executable modeling language. *Proc. 21st Annual Hawaii Int. Conf. System Sciences*, Vol. III, pp. 403–10.

CLEMENCE, JR, R. D. 1984 LEXICON: A structured modeling system for optimization. Master's Thesis, Naval Postgraduate School, Monterey, CA, USA.

CLEMENCE, JR, R. D. 1987 A type calculus for executable modeling languages. Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, USA (in preparation).

DAY, R. E., & WILLIAMS, H. P. 1986 MAGIC: The design and use of an interactive modeling language for mathematical programming. *IMA J. Math. Mgmt.* **1,** 53–65.

DOLK, D. R. 1986 Data as models: An approach to implementing model management. *Decision Support Syst.* **2,** 73–80.

DOLK, D. R. 1986 A generalized model management system for mathematical programming. *ACM Trans. Math. Software* **12,** 92–125.

DOLK, D. R., & KONSYNSKI, B. 1985 Model management in organizations. *Inf. Mgmt.* **9,** 35–47.

FOURER, R. 1983 Modeling languages versus matrix generators for linear programming. *ACM Trans. Math. Software* **9,** 143–83.

FOURER, R., GAY, D. M., & KERNIGHAN, B. W. 1987 AMPL: A mathematical programming language. Computing Science Technical Report No. 133, AT&T Bell Laboratories, Murray Hill, NJ, USA.

GEOFFRION, A. M. 1986 Structured modeling (unpublished manuscript).

GEOFFRION, A. M. 1987 An introduction to structured modeling. *Mgmt. Sci.* **33,** 547–88.

KENDRICK, D. A., & MEERAUS, A. 1987 *GAMS: An Introduction.* Palo Alto, CA: The Scientific Press.

LUCAS, C., & MITRA, G. 1988 Computer assisted mathematical programming (modeling system: CAMPS). *Comput. J.* (to appear).